

Deeksha Juneja and Michael Rhodas | ComS 319 | December 12, 2015

EMAIL APPLICATION

With ANGular JS

Table of Contents

[Overview 1](#_Toc437809122)

[Introduction to AngularJS 1](#_Toc437809123)

[Interaction with the Material 2](#_Toc437809124)

[Data Binding 2](#_Toc437809125)

[Modules 2](#_Toc437809126)

[Directives 3](#_Toc437809127)

[Controllers 4](#_Toc437809128)

[Use Strict 5](#_Toc437809129)

[View 5](#_Toc437809130)

[JSON 5](#_Toc437809131)

[Complex Topics 6](#_Toc437809132)

[Routing 6](#_Toc437809133)

[Organization 6](#_Toc437809134)

[Sugar Methods 7](#_Toc437809135)

[Changing CSS 7](#_Toc437809136)

[Adding Compose Functionality 8](#_Toc437809137)

[Factories 8](#_Toc437809138)

[Promises 9](#_Toc437809139)

[Sanitize library 10](#_Toc437809140)

[Blooms Taxonomy 10](#_Toc437809141)

[Creating 10](#_Toc437809142)

[Evalutating 11](#_Toc437809143)

[Analysing 11](#_Toc437809144)

[Applying 12](#_Toc437809145)

[Conclusion 13](#_Toc437809146)

# Overview

In our application, we are making a clone for an EMAIL application. The purpose of this application is to explore Angular JS and find out how it works.  It is a very useful JavaScript library that combines the power of data binding and elements from the MVC pattern.  With AngularJS, we can produce fully functioning applications that would take a lot of time and effort to develop than without the library.

The single page application concept brought on by Angular JS was quite intriguing to us.  To fully grasp Angular JS, a lot of learning had to be done online to think of an idea that fits what Angular JS can do for us.  Since the library is so focused on this singular app concept, we needed an idea that involved changing its view.

We explored a basic email application to understand the concepts presented by Angular JS.  We have an inbox that uses JSON files linked up to the view through Angular JS’s modules.  There is also an email reading view that takes more information binding to the view.  We have a pop up to compose an email as well.  All of this could be linked to a back end for a simply functioning email application.

This document will introduce what is AngularJS, our understanding of the concepts of AngularJS, some of the complex AngularJS components and in the end will present our learning process through the blooms taxonomy.

# Introduction to AngularJS

AngularJS is a JavaScript framework. It is basically a JavaScript library. The advantage of using AngularJS over other libraries is its ability to extend HTML. It has ng-directives which extend the capabilities of HTML throughout our application. Some of the common AngularJS directives are ng-app, ng-model and ng-bind. Our understanding of these is explored thorough out the application and is explained in the Directives section below. Moreover, AngularJS is based on the MVC pattern. MVC stands for model, view and controller respectively. The model is the data, the view is the HTML and CSS part which the user interacts with, and controller makes everything work together. AngularJS allows for Single page applications.

# Interaction with the Material

## Data Binding

One of the most important aspects of AngularJS is the ability to easily bind data in a two directional sense. The three main components in the data binding are the Model, Template, and View. A template is a bunch of HTML code that is ready to be inserted as the view with the given data from the Model. Without the AngularJS library, one would normally have to use the Template and Model to create the View. With AngularJS, the model is hooked up in a two way bind with the view components to make the model more powerful. This way, if the Model changes, the View is also automatically updated for the user.

The other bind is with the data the user enters into the View. This means that not only is the view automatically updated for the user, but the data for the program is also automatically updated when the user enters data. No extra code has to be written for this because AngularJS takes care of that for us. This extra code would be to sync the View with the Model and the Model with the View. This is all pre written in the library for us.

In Angular, the template is taken as un-compiled HTML code, from a file kind of like a string. When this template is grabbed, data is inserted into it and it becomes the View for the user. AngularJS has a labeling system which includes directives, modules, controllers, scopes, factories, and other topics to be discussed. By using these tools, the code in the background will link your View and Model. Then data changed in the View is changed immediately in the Model and also changes in the Model are reflected in the View.

## Modules

Modules are a very important part of AngularJS. Modules are the components of AngularJS that we specify to link our model with our view. Examples of modules are the different parts of our app essentially. These parts are wired together for us when normally we would have to wire them together our self when we start the program.

We can make custom directives in angular which will be used to insert views in our application. There is an overall app name declared in our HTML tag that says ng-app=’’EmailApp.” This will be what we connect our modules through.

Whenever we bootstrap another module onto our app, we must list the dependencies required so Angular can define all the functions used. For example, we use ng-route and ng-sanitize in our Angular JS application, so in our dependency array, we put those two strings in. By setting up these dependencies, we can create organization to our three Model, View, and Controller concepts. Declaring a dependency means that the module needs that declared module in order to work and fulfill its duty.

## Directives

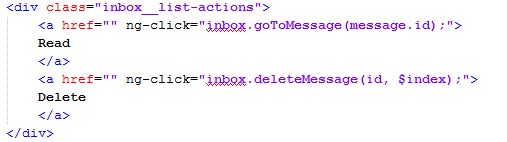
Directives are AngularJS’s capabilities to extend the HTML. Directives are written inside the HTML tags with a prefix of ng-. To include HTML validation into the application, it is always a good idea to use the directives as data-ng- instead of ng-. To name a few directives, we have used ng-app in our application which specifics the namespace of the application. Our application is called the “EmailApp”. So we extend our HTML tag to specify the ng-app like so: -



The interesting thing about ng-app name spacing is that we can give names to anything divisions and easily put separate controllers on them. Moreover, in our application we are using ng-view attribute to tell Angular where it should inject HTML. The view is injected based on the URL that a user visits. We are using a single application view.



What this does is that, when the user visits the inbox, the inbox view gets injected inside the ng-view. Furthermore, the controller associated with the inbox which is called the InboxCtrl is invoked. Some of the other directives used throughout our application include ng-model and ng-bind. Ng-model bind the values of HTML controls such as input, textarea, etc. to the application data while the ng-bind is used for binding the application data to the HTML view. In a way they work opposite to each other. We are also using ng-click throughout our application as a replacement for buttons. It is easier to associate the module with ng-clicks than make a function in JavaScript for each button. An example of that would be:-



One of the most important parts of our application is searching through the list of emails with sender’s name. If someone types into the search bar, it sets the inbox.search properly and updates the ng-repeat’s filter.

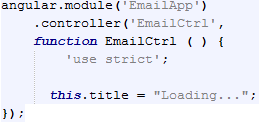


Angular has a convention of naming the directives as well. The directives are “camelCased” for example (myDirective) whereas the html element is “hyphen-separated” (<my-Directive></my-Directive>).

To simply define our understanding of directives, they are Angular’s way of customizing the HTML elements. They offer reusable way to encapsulate data and behaviors.

## Controllers

Controllers act like a middleman between our Model and our Views. Controllers make the changes in the Model and the View happen. Let us try to understand controllers through an example. If we give controller html code from the route and a JavaScript object from the dependency injection; then the controller will be able to tell the html (the view) what it can do by giving it some scope variables and functions (the JavaScript). A good controller should have as little logic as possible as they only do two things: bind the model to the view i.e. initialize the view and add the helper function to the view. Our controllers are build keeping in mind the same requirements of the controllers. We have many controllers the InboxCtrl and the EmailCtrl. The email controls for updating the side panel dynamically looks like:-



This is the complete functionality of a controller. It is important to remember that the controller is associated with the ng-controller directive.

## Use Strict

Use Strict is a new directive of JavaScript. It is a literal which is new to JavaScript 1.8.5. It helps in executing the code in a strict mode which means that it doesn’t allow for any undeclared variables to be used. It is an important security feature for JavaScript. As JavaScript is loosely typed, a bad syntax would not generally throw an error. For example, using a variable without declaring it would cause the variable to become a global variable. To prevent this and many other such cases, we have used we have used “use strict”. “use strict” acts like a great debugging tool. It has to be put at the beginning of the function to be recognized. In strict mode, any assignment to a non-writable property, a getter-only property, a non-existing property, a non-existing variable, or a non-existing object, will throw an error.

## View

The ng-view tag is used to label a point where all this HTML code can be injected into the DOM for the user to see. If we open up the console and observe the HTML code under the ng-view div tag, we will see the inbox or email view inserted there.

The AngularJS files depend on this tag so the library knows where to insert the HTML into the document for the user to see. This HTML is switched out depending on what the module is telling the view to display. There are templates of HTML made, and then the inner workings of the modules will display the correct HTML for the user.

## JSON

JSON notation, or JavaScript Object Notation, is a way to store an object in a unique way. This way, if we have a complicated object, we can send it around to our functions with ease. We can also define a separate .JSON file to use in multiple scripts if we do not want to hard code it.

We will have the prototype names defined in the JSON variable for the object, and then we can assign values to those object attributes. For this application, JSON files were used to create the fake emails in the Inbox and format the Views. It will hold data from those emails like who sent it, when it was sent, the message content, etc. This data is retrieved from the files and is used to display the different messages.

# Complex Topics

## Routing

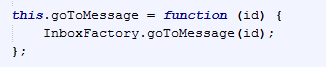
Routing is important in this application because it is used to change the view for the user. When using this style of an AngularJS app, we need include the angular-route library along with the main AngularJS library.

This controls what the view shows by using templates and the URL to route to the correct view from the model. In the project, we specify two different main routes to use, they are the inbox view and the email view. When the route is in the URL, the template is set as well as the controller. This is done by using the when() method from $routeProvider.

The template URLs are segments of html code stored in files. This html code is taken from these files and is inserted into the DOM when that route is enabled. The controller is also hooked up and changed depending on what route, or view you are in. So when in the inbox, the inbox controller is controlling the model, when in an email, the email controller is enabled. This allows us to dynamically assign a controller to a template for the user to see and manipulate the model through its elements.

The way AngularJS configures the routes is by giving a callback to .config before any other function. The config method configures your view and data to your model. The $routeProvider is set up as an argument in the app.config(). This way when the route is changed, the $routeProvider will change, then the config will change, and the view will update for the user.

When we are in the inbox, then select an email, the id of the email will determine which JSON file will be loaded for the view. This route is declared in the when statement. The id is given as a prototype in the JSON file for the email as shown below:-



## Organization

With AngularJS, our code has a very nice organizational flow to it. We first have our main app which we have at the root of our /js folder. From there, add extra folders named with the type of module we linking up to the app. For example, we have custom controllers, directives, and factory modules. So in our folder with app.js, we have a folder named after each one of those modules.

This why, your code is divided into its different purposes through these folders. Since AngularJS is all about binding our data for us by using these modules, it is good to keep them all organized so we can modify and expand on the application later. All of these modules depend on the main app from app.js, in our case it is named ‘Email App’.

Furthermore, we will usually have some sort of template we want to use with our modules to insert data into the DOM for the view to update. These templates can be placed in the folders along with the module. The email and inbox views are our main templates in this app so we put them with our directives which make use of them.

We use a single app approach to this program to make it easy to change the view without having to load to a completely different page. If you look at the URL when you change to different pages in the application, you will notice a pound sign with some label. We have an inbox view and an email view. This is how angular injects its HTML templates into the view of the main index.html file. Depending on how the Model is set up, the view will display the data from the template.

## Sugar Methods

We are using some sugar methods for using the $http to make a GET request to the JSON files. We are using JSON files to store the emails. We also set the default error handler for the http request by chaining a method with the promise returned by $http.get(). The ones we have used are error (fn) and success (fn). These methods are similar to then(fn) and catch(fn) but are specific to $http requests.

## Changing CSS

Cascading stylesheets or CSS is the most important formatting tool in our project. All our formatting is in style.css file. It defines the style of the headers, body, and divisions. CSS is such a powerful tool that it gave us the opportunity to try many different things with our EMAIL. We defined a separate color for each of our divisions were able to keep them separate by creating a different class for each division. By creating different classes, CSS allows us to modify each division separately. Hence, if there are multiple sub classes in a class, the properties of top level class will be applied to each of the sub classes. Now, sub classes can either keep the properties inherited by the top level class or over-ride them.

One of the most interesting things we did with CSS is using the hover functionality. When the user will hover over any ng-click/button, the color of the button will change showing that it is clickable. All our buttons are also associated with some functionality which enhances the user experience. A new functionality we explored was flex. The flex attribute defines the length of an item relative to all the flexible items in the container. We are using flex to keep all our views flexible. Our colors are defined be easy on eyes and the black header on top adds to the creativity of the application.

## Adding Compose Functionality

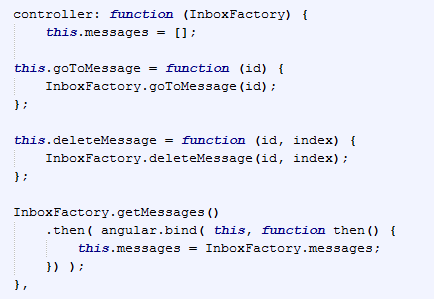
To add the compose functionality, we had a vision of wanting to create a pop up window for composing an email. This might sound easy and looks every easy as done in the code. But actually doing it took a lot of experimentation. New window could have been added through a lot of ways. Though each way came with its own problems. I am not going to claim that the method we used was the best method, but frankly that method was the only method that worked without errors. Most of the time when I tried to do it through AngularJS, instead of creating a popup window, it would create a view just below compose email in the side panel. To get rid of that, I added the whole functionality compose method in a separate file called compose.html. I am invoking it through the onClick method instead of the conventionally ng-click provided by AngularJS.

## Factories

Factories are used to help create some sort of service for the developer. A factory is a type of service that can be defined as .service instead of .factory. The service defines a function that is used to serve another part of the application. The reason we want to use a factory instead of a service here is because we want to actually create a literal object. This object is used to create the view.

Once we create this factory, we can use it in the other modules. In the email application, the factory for the inbox will gather up the date from the JSON email files and story them into an array.

We also define functions to use on this data. For example, the array of emails has a deleteMessage function. This will take the index of the email desired to be deleted and remove it from view by updating the data array. Then the new array is returned for the view.



## Promises

Promises allow us to organize functions which take a long time to execute. An example of such functions would be HTTP requests. These promises are implemented with $q. The functionality of the promises can be best described with a problem and three cases. So for example:-

Do something after the execution of the HTTP request. The promise will execute the “something” based on the result the HTTP request.

Hence if the HTTP request was completed successfully then success function will be executed. If the request is not completed successfully then then catch function gets executed.

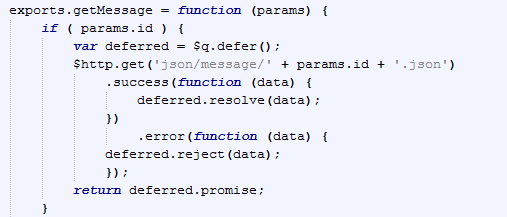
I understood this better through a real life example which I am stating below.

Every Sunday Morning, a father would as his sun to go to the hill top to find out the weather forecast for the afternoon. The son promises his dad that we will bring back the weather forecast and leaves.

Now the dad decides, if the weather is good, he will prepare for a finishing trip, if it’s bad he will stay in. Moreover, if the son is unable to bring back the weather forecast or in other words fails, he would still stay in.

What I gained from this was that son is a service and the dad is controlling the logic in this situation. The dad does other things while the son performs the service. After the son has finished service the father can either prepare for the finishing trip or stay in. The best thing about promises is that, it doesn’t block the dad from doing something else while the son gets the weather forecast.

We have used promises in both our factories for the inbox and the email. The email promise looks like: -



Here we are doing a resolve data on success and reject data on error. This means that if id matches to the message, we open the message else we don’t.

## Sanitize library

The sanitize library is used as an extension to the Angular JS to add extra functionality and security. First, we must remember to add the dependency on this library in our app.js by adding the “ng-sanitize” string to your dependency array. Then remember to include the library after our AngularJS library to load in the functions to use.

The purpose of this library is to sanitize our HTML content received from a client. By adding this library, we can use the ng-bind-html attribute. This will validate our html file and report any errors that may occur from the HTML it fetches in that tag.

# Blooms Taxonomy

## Creating

The idea was to create an email application to portray the functionality of AngularJS. Our first idea was to make a dynamic view map with AngularJS but then we decided to do a bigger experiment and make the email application. Being unfamiliar with AngularJS and having limited exposure to JavaScript posed a great challenge. We set our goals in steps. Our first goal was to create an inbox view. After we accomplished that, we decided to exploit the single page application properly of AngularJS and added a second view that is the Email view. In the email view, all the emails that populate the inbox can be read by clicking on “Read”. Once we had that accomplished, we came back to adding a new popup window for composing the email. The goal was to keep the look of the application neat and easy for the users.

## Evalutating

We had to evaluate each step we took in the design of the email application carefully. We spent a day to understand the working of AngularJS itself before we started implementing it. We assessed the interaction between the model and view. We were not used to making JavaScript and html code modular. Hence it took a little understanding on our part to have separate files for directives. So the html is in a different file and the AngularJS controller is in a different file. Hence, one of the things we really appreciate about AngularJS is the ease of linking everything and easy organization. Our application gets launched through the app.js in the /js folder which provides the route that our application will take. Another thing we learnt while creating this application was using lists in HTML. We had not experiment with using lists before and hence has to spend quite a bit of time understanding how HTML lists work. One of the most important parts of evaluation was the styling of the application. It might seem like a trivial issue but if the look and feel of an application is not appropriate then a great functioning application does not get enough users. Hence, we spent quite a lot of time designing the color scheme and overall look of the application. The greyish-brown colors are easily on the eyes of the user.

## Analysing

The analysis came in when we had to decide how to divide the work amongst our self. We both were very pumped up about using AngularJS. After we learnt AngularJS in depth together, we decided to jump right in to the application. We both started working on it simultaneously and exchanged our progress with each other.

It was important for us to analyze how AngularJS works because a lot of the functionality provided by AngularJS is highly abstract. Being developers our self, we couldn’t use something without understanding it. Though, it is important to start creating something to actually understand. After going over quite a few resources, we decided to dive into the designing process to actually understand the insides of AngularJS.

We didn’t start with using AngularJS with the email application because that seemed like a very big risk. First we made very small applications with AngularJS and explored its dynamic capabilities and after that we started to delve into the Email Application. One of the most important thing we had to analyze for this portfolio was the need to compose. We were not hooking up a real backend for our application. Hence we were pretty unsure on whether we should even have a compose functionality. After much debate, we decided to include it because even though it doesn’t actually send an email, it is the most important role of an email application.

## Applying

Application of all aspects of the project put together took us much more time than we thought it would take us. We messed up a lot of times while we were making the application. A lot of times it was hard to figure out the mistakes we had made which forced us to use “use strict”. But other than that, it was smooth sailing. This project gave us a chance to use our knowledge of JavaScript and HTML, and in turn it extended our knowledge of JavaScript, AngularJS, CSS and HTML.

One of the trickiest applications was to make the three different divisions of the webpage in different orientation.

The first step was create the inbox view for our application. That is the first thing we did because we wanted at least something to be working. We wrote the HTML for that which contained a list of emails. We later improvised it by importing the emails from JSON. The second step was to add a search feature in the email by the sender’s name. Deleting messages was a little more difficult because that had to be done via factories but we did not want the delete to be permanent either. A lot of brainstorming went into that.

After we were done with a basic inbox view, we decided to add the side division where it says compose and inbox. Having made the inbox, using AngularJS to do that was not too hard. We just had to make a view and a controller for that. Also, because the controller was not doing too much, we were rather tempted to add everything to the same file but later changed it to keep the code modular.

At this point we were rather happy with our application procedure and were making quite a lot of progress and decided to have a feature of opening the email as well. At this point we added the read in the inbox view. This was linked to opening the email through the idea provided in the JSON file. We made a separate JSON file for every email.

We followed a similar procedure for making the email portion as we did with the Inbox. We added the functionality of replying to the email and then connected it back to the inbox view via the back button. The back button had a lot of trouble. It required us to put a direct link to the index.html to work. Our side panel gets updated with the name of the email as well.

Finally, we added the hardest part with the smallest amount of code which was the compose email part. It creates a new popup window. It is a nice touch to the overall functionality of the application.

Since AngularJS is such a powerful library, it took a lot of analysis of the library to figure out exactly what was going on behind the sense. A lot of things happen behind the scenes and we were rather confused on how things are working at first. Having to revisit and analyze resources online was a part of the learning process. Then to build the application our selves took applying all that knowledge we had gathered.

# Conclusion

In conclusion, the Angular JS library is an extremely powerful tool with many different components.  The library’s genius helps developers create code that can use a MVC pattern in a unique and efficient way.  Behind the scenes, this library will link up your model to a view that will dynamically change with the data that changes.

This library helps synergize the Model with the View by being able to create custom controllers, directives, services, and so much more to your View.  The email application created for the purpose of portfolio 3 would have taken a lot more effort without this library because as the developers, we would have to write the behind the scenes code that wires the Model to the View.  This can be quite a difficult task, but the way Angular JS approaches the situation is very understandable.  After learning the syntax and how it works, we could really see how beneficial the library is.

In the end, we were able to produce an email application which has an inbox view and an email view. We also have a pop up for compose email. In our email view, we have a reply section as well. AngularJS made these tasks much simpler.